Programmation Orientée Objet : Constructeurs/Destructeurs (résumé) Compilation séparée

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

Vidéos, transparents et quiz

https://www.coursera.org/learn/programmation-orientee-objet-cpp/home/week/2

☞ Semaine 2

Construction/Destruction

Méthode **constructeur**(initialisation des attributs) :

```
NomClass(liste_arguments)
: attribut1(...), /* bloc optionel:
  ....              appels aux constructeurs
  attributN(...)    des attributs */
{
 // autres opérations
}
```

Méthode **constructeur de copie**:

```
NomClasse(const NomClasse& obj)
:...
{...}
```

Méthode **destructeur**(ne peut être surchargée) :

```
~NomClasse() {
  // opérations (de libération)
}
```

Des versions par défaut (minimales) de ces méthodes sont générées automatiquement par C++ si on ne les fournit pas

Règle: si on en définit une explicitement, il vaut mieux toutes les redéfinir !

Approche modulaire

Jusqu'à maintenant vos programmes étaient écrits en une seule fois, dans un seul fichier.

Cette approche n'est pas réaliste pour des programmes plus conséquents, qui nécessitent partage de composants, maintenance séparée, réutilisation, ...

On préfére une **approche modulaire**

c'est-à-dire une approche qui
décompose la tâche à résoudre en sous-tâches
implémentées sous la forme de **modules génériques** (qui pourront être **réutilisés** dans d'autres contextes).

Chaque module correspond alors à une tâche ponctuelle, à un ensemble cohérent de données, à un concept de base, etc...

Réutilisabilité

La conception d'un programme doit tenir compte de deux aspects importants :

- la **réutilisation** des objets/fonctions existants : bibliothèques logicielles (« libraries » en anglais);
  (les autres/passé $\longrightarrow$ nous/présent)
- la **réutilisabilité** des objets/fonctions nouvellement créés.
  (nous/présent $\longrightarrow$ les autres/futur)

Remarque : vous pouvez vous-même créer vos **propres bibliothèques**.
(mais non abordé dans ce cours)

Conception modulaire

Concrètement, cela signifie que les types, structures de données et fonctions correspondant à un « concept de base » seront **regroupés dans un fichier** qui leur est propre.

Par exemple, on définira la classe `Bacterium` dans un fichier à part.

- ☞ séparation des déclarations des objets de leur utilisation effective (dans un `main()`).

Concrètement, cela crée donc plusieurs fichiers séparés, qu'il faudra regrouper (« lier ») pour faire un programme.

Compilation séparée

☞ Pouquoi faire cela ?

- Pour rendre **réutilisable** (exemple `Vec2d`) : évite de réinventer la roue à chaque fois
- Pour **maintenir** plus **facilement** : pas besoin de tout recompiler le jour où on corrige une erreur dans `Vec2d`
- Pour pouvoir **développer** des programmes **indépendamment**, c'est-à-dire même si le code source n'est pas disponible
- **Distribuer des bibliothèques** logicielles (morceaux de code) sans en donner les codes sources (protection intellectuelle).

☞ Mais comment alors faire un tout (un programme complet) ? Comment `main()` connait-il le reste ?

Compilation séparée

La partie déclaration est la partie **visible** du module que l'on écrit, qui va permettre son utilisation (et donc sa réutilisation).

C'est elle qui est utile aux autres fichiers pour utiliser les objets déclarés.

La partie définition est l'implémentation du code correspondant et n'est pas directement nécessaire pour l'utilisateur du module. Elle peut être **cachée** (aux autres).

Compilation séparée

De ce fait, il est nécessaire (en conception modulaire) de séparer ces parties en deux fichiers :

- les fichiers de déclaration (fichiers « *headers* »), avec une extension `.h` ou `.hpp`).

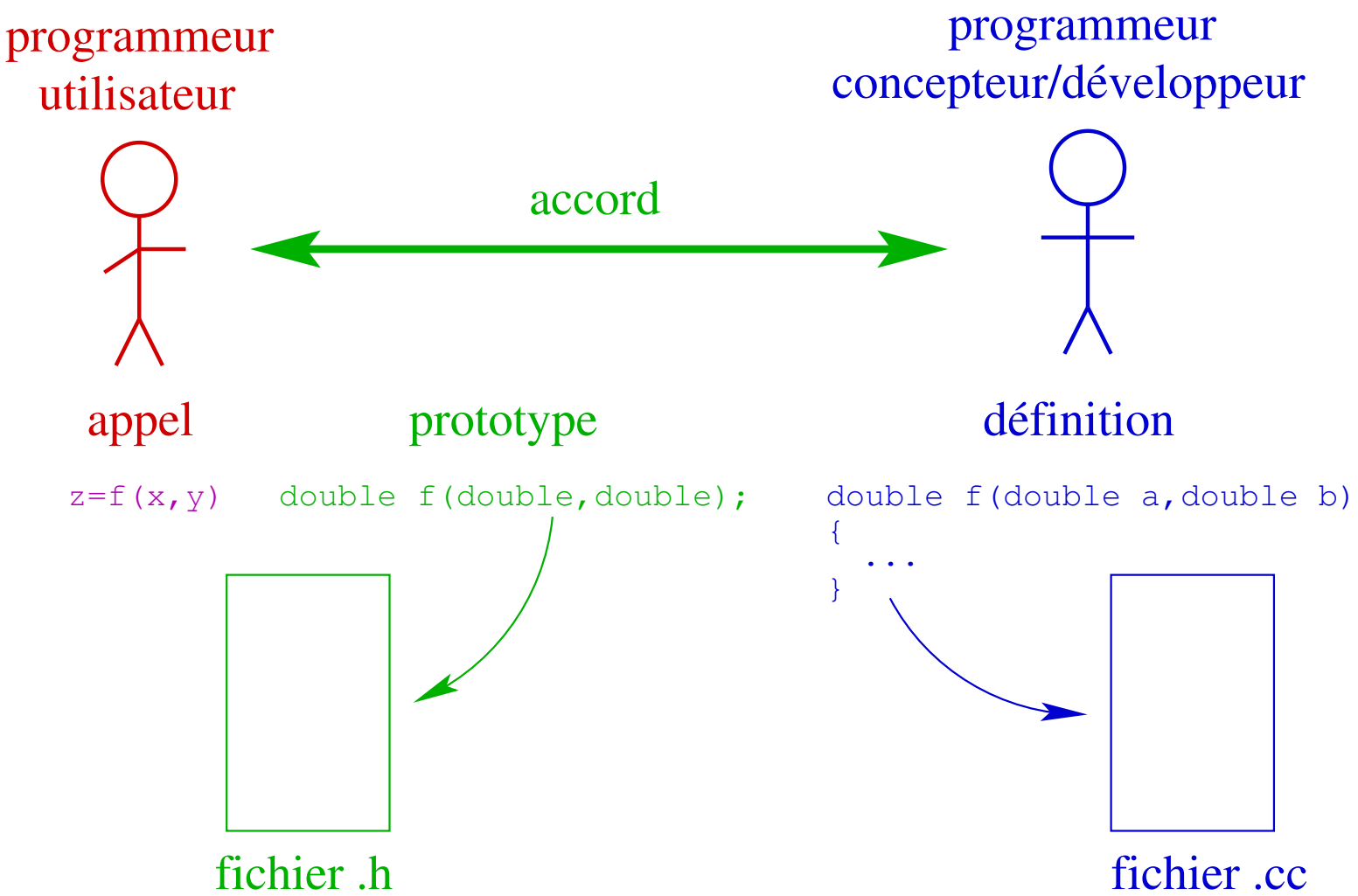
  Ce sont ces fichiers qu'on inclut en début de programme par la commande `#include`

- les fichiers de définitions (fichiers sources, avec une extension `.cc` ou `.cpp`)

  Ce sont ces fichiers que l'on compile pour créer du code exécutable.

Jamila Sam
& Jean-Cédric Chappelier

Compilation séparée



Compilation séparée (2)

La séparation des parties déclaration et définition en deux fichiers permet une **compilation séparée** du programme complet :

- phase 1 : production de fichiers binaires (appelés fichiers **objets**) correspondant à la compilation des fichiers sources (`.cc`) contenant les parties définitions (et dans lesquels on inclut (`#include`) les fichiers « *headers* » (`.h`) nécessaires) ;
- phase 2 : production du fichier exécutable final à partir des fichiers objets.

CMake

Conseils pour les inclusions

- n'inclure que ce qui est nécessaire et seulement dans le fichier où c'est nécessaire
- évitez les clauses `using namespace` dans les fichiers d'entête
- se préserver des inclusions multiples
  `#pragma once`
  au début de chaque fichier d'entête (usage pas supporté par tous les compilateurs, mais en bonne voie de généralisation), sinon:
  ```
  #ifndef POOSV_NOM_DE_LA_CLASSE
  #define POOSV_NOM_DE_LA_CLASSE
  ```
  suivi de `#endif`
  à la fin du fichier d'entête.
- si les types utilisés sont des **pointeurs**, préférez la prédéclaration du type à l'inclusion du fichier `.h/.hpp` correspondant dans le fichier d'entête. Le `.h/.hpp` sera alors plutôt inclus dans le fichier de définition

Pour préparer le prochain cours

- Vidéos et quiz du MOOC semaine 3 :
  - Variables et méthodes de classe [13:10]
  - Surcharge d'opérateurs : introduction [11:09]
  - Surcharge d'opérateurs : surcharge externe [17:28]
  - Surcharge d'opérateurs : surcharge interne [12:08]
  - Surcharge d'opérateurs : compléments [23:09]

- Le prochain cours :
  - de 14h15 à 15h (résumé et quelques approfondissements)